A Primary Study of Bug Patterns in Quantum Programs

Pengzhan Zhao
Kyushu University

Jianjun Zhao
Kyushu University

Lei Ma
Kyushu University

ABSTRACT

Bug patterns are erroneous code idioms or bad coding practices that have been proved fail time and time again, usually caused by the misunderstanding of program language features, the use of erroneous design patterns, or simple mistakes sharing common behaviors. This paper presents and categorizes a taxonomy of bug patterns in the quantum programming language Qiskit and discuss how the existing techniques can be applied to eliminate or prevent those bug patterns. We take this research as the first step to provide an underlying basis for testing and debugging of quantum programs.

KEYWORDS

Quantum program debugging, quantum bug patterns, Qiskit

1 INTRODUCTION

In modern software development, software debugging and testing are critical parts of an integrated software development method. An appropriate method of bug finding can quickly help developers locate and remove bugs. A software bug is regarded as the abnormal program behaviors which deviate from its specification [3], including poor performance when a threshold level of performance is included as part of the specification. Bug patterns are recurring relationships between potential bugs and explicit errors in a program; they are common coding practices that share similar symptoms and have been proven to fail time and time again. Those bug patterns are raised from the misunderstanding of language features, the misuse of positive design patterns, or simple mistakes having common behaviors. Such bug patterns are an essential complement to the traditional design patterns [6], just as a good programmer needs to know design patterns which can be applied in various context and improve the software quality, also to be a good software developer or problem-solver, the knowledge of common causes of faults is a need in order to know how to fix the software bugs.

In the previous research of bug patterns, most of the work focused on classical programming languages such as Java. Allen [3] summarizes more than 14 categories of bug patterns in Java, and the FindBugs research group identified more bug pattern classifications at the University of Maryland [9]. Farchi [5] also presents some concurrent bug patterns and discusses how to detect them.

Quantum programming is the process of designing and building executable quantum computer programs to achieve a particular computing result and is drawing increasing attention recently. A number of quantum programming approaches are available to write quantum programs, for instance, Qiskit [17], Q# [19], ProjectQ [20], Scaffold [1], and Quipper[7]. However, the current research so far in quantum programming is focused on problem analysis, language design, and implementation. Even though program debugging and software testing are important, it has received little attention in the quantum programming paradigm. The new complexity introduced in quantum programming makes it difficult to find the defects or bugs in the source code. Until now, only a few approaches have been proposed for testing and debugging quantum software [8, 11, 13, 14] and none of the previous work was focused on the bug pattern identification in a quantum programming language. The testing and debugging issues remain a big problem for quantum programs.

We may not know what types of bugs are unique or common happened to quantum programs without a proper bug pattern classification, and this poses several restrictions on the research and development of programs in the language:

- Developers do not know what kind of bugs are most likely to happen in a program, and therefore do not know how to prevent them. In other words, a programmer would lack a piece of fundamental knowledge on how to write bug-free code.
- Testers do not have sufficient knowledge of how to write adequacy test cases that can effectively cover most common potential errors. Only when having an idea of how the common bugs happened in programs can the tester set up criteria for better addressing the specific bugs.
- Software maintenance staff do not know which features of the language are more likely to result in the incorrect code; so they cannot clearly view the current system when doing the maintenance tasks.

The bug patterns may help to solve these problems. To identify such patterns in a quantum programming language explicitly, we can leverage many programmers' experience to improve their productivity in bug finding and eliminate the cost of software maintenance. The bug pattern identification can also help language designers or tool developers develop the corresponding bug finding techniques or bug detectors, which could be applied to locate bugs in the source code by program analysis.

Furthermore, the bug patterns provide a basis for further research on debugging quantum programs. It provides insight into the possible consequences of different bug types and summarizes the common behaviors among similar ones. It can be used to recognize faults that have been already existed and prevent potential bugs. The bug patterns taxonomy for quantum programming languages such as Qiskit could be seen as a starting point for creating the general bug patterns for quantum programming languages.

This paper chooses the widely used quantum programming language Qiskit as our target language and identifies the common bug patterns. We also show the corresponding example for each bug pattern to illustrate these patterns' symptoms. To the best of our knowledge, the work described in this paper is the first attempt to identify the bug patterns existing in Qiskit programs systematically.

The rest of the paper is organized as follows. Section 2 briefly introduces the background knowledge of quantum programming in Qiskit and the new error-prone features introduced by quantum programs. Section 3 describes the identified bug patterns in Qiskit in detail. Related work is discussed in section 4 and concluding remarks are given in Section 5.

```
simulator = Aer.get_backend('qasm_simulator')

qreg = QuantumRegister(3)
creg = ClassicalRegister(3)
circuit = QuantumCircuit(qreg, creg)

circuit.h(0)
circuit.h(2)

circuit.cx(0, 1)

circuit.measure([0,1,2], [0,1,2])

job = execute(circuit, simulator, shots=1000)
result = job.result()
counts = result.get_counts(circuit)
print(counts)
```

Figure 1: A simple quantum program in Qiskit

2 BACKGROUND

We next briefly introduce the background information on programming in Qiskit and the error-prone features in Qiskit programs.

2.1 Qiskit

Qiskit is a Python-based quantum programming language that allows us to create algorithms for a quantum computer [12]. As a Python package which provides tools for creating and manipulating quantum programs and running them on prototype quantum devices and simulators [2], Qiskit is one of the most widely used open-source frameworks for quantum computing. It can use the built-in modules for noise characterization and circuit optimization, which is useful for research in reducing the effects of noise. Qiskit also has its library of quantum algorithms for machine learning, optimization, and chemistry.

In Qiskit, an experiment is defined by a quantum object data structure that contains configuration information and the experiment sequences. The object can be used to get status information and to retrieve results [16]. Figure 1 shows a simple Qiskit program, which performs an example of the entire workflow of a quantum program. The function`Aer.get_backend('qasm_simulator')` returns a backend object for the given backend name(`qasm_simulator`). The `backend` class is an interface to the simulator. And the actual name of `Aer` for this class is `AerProvider`. After the experimental design is completed, run the instructions through `execute` method. The `shots` of the simulation which means the number of times the circuit is run, was set to be 1000 while the default is 1024. When we want to output the results of a measurement, the "job" will have a method `job.result()` to retrieve measurement results. Then, we can access the counts via the method `get_counts(circuit)` which gives the aggregate outcomes of the experiment.

2.2 The Properties of Qubits

In the following, we use Qiskit as an example to explain the characteristics of quantum bit (qubit) and the necessary execution process of a complete quantum program.

In quantum computing, the basic unit of information is the qubit. As shown in Figure 1, `qreg = QuantumRegister(3)` means assigning a quantum register of three qubits, and the value of each qubit is $|0\rangle$ by default. So the initial value of these three qubits is $|000\rangle$. Next, let the first and third qubits pass through the H (Hadamard) gate. As shown by `circuit.h(0)` and `circuit.h(2)`. In this way, the unique property "superposition" of qubits is realized. It means each qubit can take on values of $|0\rangle$ and $|1\rangle$. There is also an "entanglement" of qubit properties that only multiple qubits can achieve. The code in the sample program is `circuit.cx(0,1)`. That is to say; the first qubit is entangled with the second qubit through a CNOT (Controlled-NOT) gate operation. We measure the first qubit, and its output is `0` for 50 percent probability and `1` for 50 percent probability. After that, measuring the second qubit is 100 percent the same as the first measurement result. The measurement statement of qubits shown in Figure 1 is `circuit.measure([0,1,2], [0,1,2])`. Measurement will lead to the collapse of the quantum superposition state, which will eventually be a classical state of measurement. There are many kinds of quantum measurements, and the projection measurement of a single qubit is used here. That is, each qubit is projected onto a state space consisting of base vectors $|0\rangle$ or $|1\rangle$. In this program, the final output is a three-bit array.

2.3 Error-Prone Features in Qiskit Programs

By focusing on the language features of Qiskit, we can classify the bug patterns in Qiskit into the following four categories.

- **Initialization**: A quantum program is a series of operations on qubits. The initial stage is to initialize the quantum registers to store the qubits that need to be manipulated. Then the classical registers are initialized to store the values of the measured qubits. This stage does not include setting the quantum state, as the quantum state setting needs to be implemented by a gate operation. Quantum registers and classical registers do not have to be of the equal initial size. When we use multiple classical bits to store the same qubit measurements, we need to initialize as many classical bits as possible. However, another case is that the initialized qubit is larger than the classical bit. Since the programmer does not intend to measure some qubits, it is assumed that there is no need to initialize the classical bits equal to the qubits. Nevertheless, this is also the reason why most programs go wrong. So a hasty initialization can cause some problems for subsequent programs.
- **Gate Operation**: The core of quantum computing is to operate on qubits. Qiskit provides almost all the gates to implement algorithms in quantum programs [18]. To achieve the "superposition" of qubits, it must pass through the H (Hadamard) Gate. To achieve "entanglement" in the case of multiple qubits, it must pass through the CNOT (controlled-NOT) gate. In quantum language, complex gate operations are decomposed into basic gates and gradually realized. Controlled gates are parameterized by two qubits, and double-controlled gates require three qubits. However, this does not mean that the double-controlled gate operates on three qubits at the same time. Many errors may occur when inappropriately using gates that operate on the qubits multiple times.
- **Measurement**: When we want to obtain the output, we must perform a measurement operation on the target qubit. The measured qubit is returned as the classical state's value, which no longer has superposition properties. So the qubit that has been

```
qreg = QuantumRegister(3)
creg = ClassicalRegister(2)
circuit = QuantumCircuit(qreg, creg)

circuit.h(0)
circuit.cx(0, 1)
circuit.cx(1, 2)

circuit.measure([0,1,2], [0,1,2])
```

Figure 2: Unequal Classical bits and Qubits

measured cannot be used as a control qubit to entangle with other qubits. Although measurement is a simple operation, the program executing a measurement statement is very complicated. It requires thousands of projection measurements of the qubits. Finally, it outputs all its possible results. Moreover, the number of occurrences of the result is used to obtain the size of the probability of outputting the correct value. Many errors start with the measurement statement because programmers do not really understand the effect of measurements on the state of qubits.

- **Deallocation**: It is crucial to reset and release the qubits safely; otherwise, the auxiliary qubits in the entangled state will affect the output. Deallocation is not considered to be a specific operation due to the power of Qiskit. We do not need to reset the qubits manually. However, In some backends, not releasing all qubits can be problematic. In Qiskit, not handling all the qubits in the entangled state can cause problems in the program or output unsatisfactory values.

For better understanding, we propose these bug patterns in terms of the quantum program execution order, which consists of four stages (processes) that the program's qubits go through, and each stage interacts with the others.

3 BUG PATTERNS IN QISKIT

We next introduce six bug patterns in Qiskit as examples. When introducing each bug pattern, we also show an example that contains this specific pattern. Since most bug patterns have some representation variants and alternatives, we choose the one that appears to be the most generally applicable. These bug patterns are also summarized in Table 1.

3.1 Unequal Classical Bits and Qubits

In Qiskit, each classical bit in the classical register stores a measured qubit value. Therefore, it is better to initialize quantum registers of the same size as classical registers. Otherwise, the bug pattern of "*Unequal Classical Bits and Qubits*" may occur, especially when the number of qubits in the quantum register is greater than that of classical bits in the classical register. From the point of view of program integrity, every used qubit should be measured.

As shown in Figure 2, when we want to measure the third qubit, we receive an error message *CircuitError: 'Index out of range.'*. If we do not measure one of the qubits, then a qubit will not get reset.

Another case is that the number of bits in the classic register is larger than the qubit. Unless we encounter the need to use multiple classical bits to store a qubit measurement, otherwise, this is not a good habit. On the one hand, resources are wasted when the program is actually developed, and on the other hand, outputting all classic bits will cause very messy results. Therefore we do not recommend this operation.

```
from qiskit import QuantumCircuit, QuantumRegister
from qiskit.circuit import Gate
import schedule,transpyle

qc = QuantumCircuit(3,3)

gt = Gate('my_custom_gate', 3, [])

qc.h(0)
qc.sdg(0)
qc.y(1)

qc.append(gt, [0,1,2])

qc.add_calibration(gt, [0,1,2], schedule)
qc = transpile(qc, backend,
               basis_gates=['u1', 'u2', 'cx', gt])

qc.measure([0,1,2], [0,1,2])
```

Figure 3: Custom gates not recognised by Qiskit

3.2 Custom Gates not Recognised

When defining a custom gate in a program, some programmers will want to define a "basic gate" that controls more than two qubits directly; the bug pattern *Custom gates not recognised by Qiskit* may occur. This pattern refers to a custom gate that does not use the gate class provided by Qiskit correctly. Alternatively, the gate is not recognized by Qiskit.

An example of an error code is shown in Figure 3, which is a program that tends to define a three-qubit controlled gate. First define a gate named `my_custom_gate` using the `Gate` method, and control the number of qubits to three. When we call this gate, the program will have an error. Because in `basic_gates`, the custom gate `gt` is not the same as other Qiskit-based gates.

This bug pattern is mainly caused by programmers who do not really understand quantum gates. Quantum gates can only control a maximum of 2 qubits and are known as basic gates. The compound gates we usually use, such as the double-controlled gate CCX(Toffoli) [17], are not the gates that directly control three qubits. Instead, multiple single-qubit gates and controlled gates are combined, resulting in a dual controlled gate effect. The correct custom gate should be a composite gate combining the basic gates provided by Qiskit and applied to the circuit.

3.3 Parameter Misuse of Complex Gate Instructions

In quantum Experience API error codes [17], there is an error indicating that the instruction is not in the gate of the call. Since many instances of such an error are found in practical Qiskit programming, we summarized this problem as a bug pattern of "*Parameter Misuse of Complex Gate Instructions*."

As a simple example, consider the code snippet in the figure 4, which calls the CX (Controlled-x) gate and the CCX (Toffoli) gate separately. There is no problem when running the instruction `qc.cx`

```
qc = QuantumCircuit(3,3)

qc.cx(0, 1, label='Label', ctrl_state=0)
qc.ccx(0, 1, 2, label='Label', ctrl_state=1)
```

Figure 4: Parameter Misuse of Complex Gate Instructions

of the CX gate. But when executing the instruction `qc.ccx` of the CCX gate, the program will report an error of `ccx() got an unexpected keyword argument 'label'`. This error is because the complex gates in Qiskit do not support `label` or `ctrl_state` as parameters as that for basic gates. This bug pattern shows that if the programmer lacks some knowledge about double-controlled gate or other complex gates and uses the parameters from these gates as that for the basic gates, it probably leads to this kind of bug.

3.4 Multiple/Repeated Measurement

Some simulator backends are unable to execute the circuit when the measurement operation performed on the qubit is repeated too many times. Alternatively, when some methods, such as `c_if`, are called but do not give the correct result. This situation may lead to the bug pattern of "*Multiple/Repeated Measurement*."

```
def get_circuit(n):

    qreg = QuantumRegister(1)
    creg = ClassicalRegister(n)
    mreg = QuantumRegister(1)
    dreg = ClassicalRegister(1)

    circ = QuantumCircuit(qreg, mreg, creg, dreg)

    for i in range(n):
        circ.measure(qreg[0], creg[i])

    circ.x(mreg[0]).c_if(creg, 0)
    circ.measure(mreg[0], dreg[0])

    return circ

b_aer = BasicAer.get_backend('qasm_simulator')
aer = Aer.get_backend('qasm_simulator')

circ65 = get_circuit(65)

print("65clbits(Aer):", execute(circ65, aer).
      result().get_counts())
print("65clbits(Basic_Aer):", execute(circ65, b_aer).
      result().get_counts())
```

Figure 5: Multiple/Repeated Measurement

To show this bug pattern, consider the piece of code in Figure 5. This is a test used to measure quantum characteristics in a computing backend simulator repeatedly. We use the Qiskit "Aer" simulator backend and the Python-based quantum simulator module "BasicAer" to simulate the circuit `qasm_simulator`. The same qubit is used multiple times here. When we call `BasicAer`, the system may report an error that the number of qubits is greater than the maximum (24) for `qasm_simulator`. Not only that, the `c_if` method we called did not get the desired result on the "Aer" backend simulator, that is, the qubits of the `mreg` register did not achieve flipping. While

```
tq = QuantumRegister(3)
tc0 = ClassicalRegister(1)
tc1 = ClassicalRegister(1)
tc2 = ClassicalRegister(1)

teleport = QuantumCircuit(tq, tc0,tc1,tc2)
teleport.h(tq[1])
teleport.cx(tq[1], tq[2])

teleport.ry(np.pi/4,tq[0])

teleport.cx(tq[0], tq[1])
teleport.h(tq[0])
teleport.barrier()

teleport.measure(tq[0], tc0[0])
teleport.measure(tq[1], tc1[0])

teleport.cx(tq[1], tq[2])
teleport.cz(tq[0], tq[2])
teleport.measure(tq[2], tc2[0])

backend = Aer.get_backend('qasm_simulator')
job = execute(teleport, backend, shots=1, memory=True).
      result()
result = job.get_memory()[0]
print(job.get_memory()[0])
```

Figure 6: Incorrect Operations after Measurement

the code `circ.x(mreg[0]).c_if(creg,0)` did not achieve. And if `n=63` in the classic register `creg`, the system will hang.

In summary, we do not recommend excessive measurement operations on qubits. The measured qubit is placed in the first position of the quantum register, and then the measurement is placed in the second position. Such repeated operations are equivalent to operating "N" multiple qubits. As a result, it can make the system extremely unstable.

3.5 Incorrect Operations after Measurement

When the measurement is completed, we cannot use the measured qubit for "entanglement." Otherwise, we will not get the desired result. The result of the measurement can be treated as a classical value that no longer has the properties that the qubit has. If the measured value continues to be entangled with other qubits, which is used to change the target qubit state, it will be the bug pattern of "*Incorrect Operations after Measurement*."

Considering the code snippet in Figure 6 taken from GitHub document [15], which realizes a quantum teleportation protocol.

In the code, the last qubit's state should be changed according to the first two bits' measurement results. The wrong instructions in the example are `teleport.cx(tq[1],tq[2])` and `teleport.cz (tq[0],tq[2])`, which entangle the measured qubit with the unmeasured qubit, and therefore affect the result of the last qubit. This mistake is quite common, and many programmers inadvertently use measured qubits. In this program, the correct code should be `teleport.z(tq[2]).c_if(tc0,1)` as well as `teleport.x (tq[2]).c_if(tc1,1)`.

Although these erroneous operations follow quantum measurements, the reason for this lies in a poor understanding of the effect of measurement operations on qubit states.

Table 1: A catalog of bug patterns in Qiskit

Bug Patterns	Category	Symptoms	Causes	Cures and Preventions
Unequal Classical Bits and Qubits	1	Classical registers are not large enough to store the measured qubits	The initialized classical bits are smaller than the qubits used or to be measured	Try to initialize quantum and classical registers of the same size
Custom gates not recognised	2	The program is unable to customize the gate function and will often report errors	Creating gates that directly control more than three qubits does not follow the principle of two qubit entanglement	Try to use the gates provided by Qiskit for the implementation of the algorithm
Parameter Misuse of Complex Gate Instructions	2	Instructions are not in the gate being used	The wrong parameters are used in some controlled gate methods (especially complex gates)	Correct understanding of a complex gate like a dual control gate, using the correct instruction
Multiple/Repeated Measurement	3	Output error or program error when measuring the same quantum bit multiple times with a `for` loop	number of measurements repeated several times	Reduction of meaningless measurements.
Incorrect Operations after Measurement	3	Unable to get the desired post-measurement result	Continued manipulation of the qubit being measured, such as changing its state or re-entangling with other qubits	The measured result cannot be used as a condition unless it is re-operated and measured as the initial qubit after reset
Unsafely Uncomputation	4	The program reports an error or does not achieve the desired result	Auxiliary qubits are not reset and remain entangled with the target qubit, which can affect the results of the target qubit measurement	Correctly reset or release all qubits to ensure they are in their initial or post-measurement states
MCX Gate usage Issue	2	Circuit cannot load MCX gates. MCX is not a Qiskit-provided basic gate that can be called directly	Although Qiskit provides MCX gate, it also provides the corresponding methods, properties, and parameters. However, this is a custom gate, which cannot be called directly	This gate needs to be customized in OpenQASM. MCX gates do exist, but the exact number of qubits to operate and how to operate them are all needed write manually
Insufficient Initial Qubits	1	Causes VQE not to respect the form of input variables and outputs the wrong circuit	When the `TwoLocal` method is used, a dual-local parametric circuit consisting of alternating rotating and entangled layers can be formed	When using parametric circuits or methods involving quantum "entanglement," initialize the values supported by the parameter `num_qubits`.
Inappropriately Modification of Register Size	1	Changing the register size may cause the program to report an error. Especially for building complex circuits	Changing the size of a register may change the hash value of the register and its bits, thus prohibiting it from being used as a key for structures such as sets	It is possible to reinitialize the registers. Otherwise, it is not recommended to modify the values of the registers without changing the variable names
Method `measure_all`	3	The program outputs the results of all measured qubits normally. However, it also outputs the classical register values	When the `measure_all` method is used, the program automatically creates a classical register to store all the qubits being measured	If we want to call the `measure_all` method to measure all qubits, we do not need to initialize the classical registers

3.6 Unsafely Uncomputation

Qiskit is a compelling framework because it supports the automatic management of qubits, i.e., there is no need to do the work of unallocated qubits manually. However, this is also its downside, as different backends will have their own implementations, which can lead to exceptions in different backends and the need to manually unallocated qubits. Thus the bug pattern of "*Unsafely Uncompntation*" appears. For example, even in Qiskit, exiting the program before measuring all qubits can lead to some backend simulators' errors. The errors can be "`Qubit has not been measured/uncomputed`."

3.7 A Catalog of Bug Patterns in Qiskit

Quantum programming introduces new quantum-aware bug patterns that differ from existing classical bug patterns. These quantum bug patterns should be identified, and a catalog for these patterns should be presented. Due to space limitation, however, we cannot explain more bug patterns in this paper; and in Table 1 we list the bug patterns in Qiskit we identified, including those detail described in this section. To classify the bug patterns listed, we summarize the description for each pattern by pattern name, category, symptoms, causes, and cures & preventions. Note that this is just a preliminary list of bug patterns in Qiskit, and more bug patterns will be added to the list as we get some new progress. In the current bug pattern catalog in Table 1, we classify these bug patterns by initialization (1), gate operation (2), measurement (3), and deallocation (4).

4 RELATED WORK

The previous research on bug patterns is mainly focused on object-oriented programming languages such as Java. Allen [3] summarizes more than 14 bug patterns categories in Java. Following Allen's work, Hovemeyer and Pugh [9] present a novel syntactic pattern matching approach to detecting the bug patterns in Java and implemented a bug finding tool called FindBugs [4], which uses various type of bug detectors to find bug patterns in Java. The bug patterns reported by FindBugs, are more than 300 patterns, which cover bad practices, performance issue, correctness, and multi-threaded correctness. Our work extends the bug patterns research and identification to the quantum programming languages using the Qiskit language as an example. The bug patterns presented in this paper are different in nature from the existing bug patterns in classical programming languages in the sense that they explicitly involve the quantum programming language features such as superposition, entanglement, and no-cloning.

Huang and Martonosi [10, 11] study the bug types for special quantum programs to support quantum software debugging. Based on the experiences of implementing some quantum algorithms, several types of bugs specific to quantum computing are identified. These bug types include incorrect quantum initial values, incorrect operations and transformations, the incorrect composition of operations using iteration, the incorrect composition of operations using recursion, incorrect composition of operations using mirroring, incorrect classical input parameters, and incorrect deallocation of qubits. The defense strategy for each bug type is also proposed, which mainly uses some assertions to detect the bugs in runtime. While Huang and Matonosi's work targets quantum software's debugging, which mainly involves the runtime execution of the software, our work targets identifying the bug patterns to support bug detection through static analysis, which considers more efficient than dynamic analysis.

5 CONCLUDING REMARKS

This paper has identified some bug patterns in quantum programming language Qiskit to provide both researchers and programmers a clear view of what kind of bugs may happen in quantum programs and how to detect them. The study of bug patterns mainly focuses on bug pattern symptoms, cause root, and cures and preventions. These bug patterns are the first result of our research and do not use every possible quantum related construct or cover all characteristics of a quantum programming language. New research should cover other remaining quantum-related constructs, as well as the interactions between them.

In future work, we would like to investigate more bug patterns in Qiskit programs. We would also like to develop a bug detecting tool based on the identified bug patterns to support bug finding in Qiskit programs.

REFERENCES

[1] Ali J Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. 2012. *Scaffold: Quantum programming language*. Technical Report. Department of Computer Science, Princeton University.

[2] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, D Bucher, FJ Cabrera-Hernández, J Carballo-Franquis, A Chen, CF Chen, et al. 2019. Qiskit: An open-source framework for quantum computing. *Accessed on: Mar* 16 (2019).

[3] Eric Allen. 2002. *Bug patterns in Java*. APress LP.

[4] Nathaniel Ayewah and William Pugh. 2010. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*. 241–252.

[5] Eitan Farchi, Yarden Nir, and Shmuel Ur. 2003. Concurrent bug patterns and how to test them. In *Proceedings international parallel and distributed processing symposium*. IEEE, 7–pp.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns Elements of reusable object-oriented sofware*. Addison Wesley.

[7] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 333–342.

[8] Shahin Honarvar, Mohammadreza Mousavi, and Rajagopal Nagarajan. 2020. Property-based testing of quantum programs in Q#. In *First International Workshop on Quantum Software Engineering (Q-SE 2020)*.

[9] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *Acm sigplan notices* 39, 12 (2004), 92–106.

[10] Yipeng Huang and Margaret Martonosi. 2018. QDB: From quantum algorithms towards correct quantum programs. *arXiv preprint arXiv:1811.05447* (2018).

[11] Yipeng Huang and Margaret Martonosi. 2019. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture*. 541–553.

[12] Daniel Koch, Laura Wessing, and Paul M Alsing. 2019. Introduction to Coding Quantum Algorithms: A Tutorial Series Using Pyquil. *arXiv preprint arXiv:1903.05195* (2019).

[13] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-based runtime assertions for testing and debugging Quantum programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

[14] Ji Liu, Gregory T Byrd, and Huiyang Zhou. 2020. Quantum circuits for dynamic runtime assertions in quantum computation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1017–1030.

[15] Manoel Marques. 2020. Qiskit Community Tutorials. *Accessed on: April, 2020* (2020). https://github.com/qiskit-community/qiskit-community-tutorials

[16] David C McKay, Thomas Alexander, Luciano Bello, Michael J Biercuk, Lev Bishop, Jiayin Chen, Jerry M Chow, Antonio D Córcoles, Daniel Egger, Stefan Filipp, et al. 2018. Qiskit backend specifications for OpenQASM and OpenPulse experiments. *arXiv preprint arXiv:1809.03452* (2018).

[17] IBM Research. 2017. Qiskit. *Accessed on: April, 2020* (2017). https://qiskit.org

[18] IBM Research. 2020. IBM Quantum Experience. *Accessed on: April, 2020* (2020). https://quantum-computing.ibm.com/docs/

[19] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. 1–10.

[20] ProjectQ Team. 2017. ProjectQ. *Accessed on: April, 2020* (2017). https://projectq.ch/